Quadruped Robot Locomotion with Reinforcement Learning
Learning Agent Code and Report (C&R)

GROUP 18: HO KAE BOON (A0219811H), HUANG WENBO (A0285085W), ZENG ZHENGTAO (A0285134E), ZHONG HAO (A0284914W)

With our Gym environment ("aliengo_gym.py") and neural networks ("DDPGNet.py") from the previous two assignments ready, we now move on to the learning agent (written inside the "LA_demo_code.ipynb" file). In that Jupyter Notebook, the code is written in multiple cells. The more important ones include a "Parameter Cell," which is used to define various variables, including the path to directories, training parameters, noise parameters, as well as helper functions. The next cell, the "Worker Cell," is used to define a *Worker* class that takes in the following parameters during initialization: an *AliengoEnv* class instance, a *DDPGNet* class instance, and an *OUNoise* class instance to carry out interactions between them. It has a *Worker.train()* method specifically used for the neural network backward pass and returning the actor and critic loss, and a *Worker.work()* method to perform the training or testing. The following cell, the "Main Code Cell," is the driver code where we execute the worker.

DDPG Training: *Worker.work()*

The main algorithm employed in our project is Deep Deterministic Policy Gradient (DDPG). This algorithm follows the Actor-Critic structure and is well-suited for continuous spaces. In our "Main Code Cell", we initiate a TensorFlow session and global variables. Subsequently, we create a *Worker* class instance and invoke the *Worker.work()* method, passing in our current session. The *Worker.work()* will perform the following:

We begin by iterating through episodes. For each episode, we reset our environment and noise, and clean our *episode_buffer*—a list used to store experiences (i.e., <*s* = current observation, *a* = action, *s1* = next observation, *r* = reward, *d* = done/is termination state>) at each time step. Within the episode loop, we iterate through time steps. At each time step, we perform the following sequentially:

1. Obtain the current state information by calling *env.get_state()* and scale it.
2. Obtain the action *a* by forward-passing our current observation *s* into the actor network using *sess.run(net.actor_net_output, feed_dict).*
3. Scale the action and add Ornstein-Uhlenbeck (OU) noise by calling *noise.get_action() method.*
4. Obtain a set of experiences <*s, a, s1, r, d*> by feeding the action to our Gym environment using *env.step(a)*. Rescale the experience using *scale_observation()* and *scale_action()* functions, then append them to the *episode_buffer*.
5. Check if the *episode_buffer* has reached the *EXPERIENCE_BUFFER_SIZE*. If yes, call the *Worker.train()* method, which will perform batch sampling, backward pass of the actor network and critic network to obtain the actor and critic loss, and soft update of the target network (more information below).
6. Set the next observation as the current observation.
7. Check if the termination state has been reached or if the timestep has exceeded the *MAX_EPISODE_LENGTH.* If so, break out of the timestep loop and proceed to the next episode.

DDPG Training: *Worker.train()*

When the *Worker.train()* method is called, a batch of experience data <*s, a, s1, r*> is randomly collected from the provided episode buffer, serving as mini-batch training data. After adjusting their dimensions, we initiate the backward pass of the actor network, followed by a backward pass of the critic network. Subsequently, soft updates of the target network are executed. For more detailed information on how the various networks tie into the DDPG algorithm, including the implementation of backward passes and soft updates, you may refer back to our previous Neural Network Report (ME5418_Neural_Network_Report.pdf), where we have extensively explained these aspects.

DDPG Training: Ornstein-Uhlenbeck Noise

To encourage exploration, we've implemented an *OUNoise* class inside our "DDPGNet.py" file, which introduces time-dependent noise to our actions. The *OUNoise.evolve_state()* method updates the current noise state (*self.state*) by incorporating a state change given by the equation: *dx = self.theta * (self.mu - x) + self.sigma * np.random.randn(action dimension).* This equation pulls the state back towards *self.mu* (mean value) and introduces random noise using a standard normal distribution scaled by *self.sigma* (degree of perturbation).

Meanwhile, the *OUNoise.get_action()* method takes in an action and a timestep and returns the action with added noise. This is achieved by summing the action with the current noise state, obtained by calling *OUNoise.evolve_state()*, and then clipping it within the action boundaries specified by *self.low* and *self.high*. Before returning the value, it updates *self.sigma* (degree of perturbation) for the next calculation using the equation: *self.sigma = self.max_sigma - (self.max_sigma - self.min_sigma) * min(1.0, t / self.decay_period)*, where *t* is the provided time step. This equation adjusts the noise amplitude based on the decay period and clips it within specific bounds.

DDPG Training: Tensorboard logging / Summary Writing
During the training operation, we leverage TensorBoard to visualize our training metrics and performances, including actor loss, critic loss, episode reward, and episode length. In the *Worker.work()*, we pass in a *tf.summary.FileWriter()* instance. We calculate the episode length (total steps taken) and episode reward, appending them to a list. This list, along with the actor and critic loss obtained when calling *Worker.train()*, is added to our summary file writer for every *SUMMARY_WINDOW* episodes. These summaries are then flushed to the *'/tf_summary_save'* directory. To visualize them in a web browser, run *tensorboard --logdir /path/to/log/directory* in the terminal. TensorBoard will access the summary files and display the graphs.

DDPG Testing
We have defined a boolean variable *TRAINING* to distinguish between training and testing operations. Within *Worker.work()*, when *TRAINING* is set to False, we skip the noise addition and *Worker.train()* operations. This way, we will solely perform a forward pass of the actor network using the current observation, *s* until the termination state is reached. The testing process will iterate for *NUM_EXPS* episodes.

DDPG Training & Testing: Save Model Checkpoints
In addition, we have implemented model checkpoints to facilitate the easy saving and restoration of our neural network model parameters. This functionality allows us to store model checkpoints in the *'/model_save'* directory, enabling the continuation of training across multiple runs. To save the model, we define a *tf.train.Saver()* instance in the main code cell and pass it into our worker. The worker then saves the session's model checkpoint for every *SUMMARY_WINDOW* episode, using the file name format *'model_<MODEL_NUMBER>.ckpt*'. We also introduce a boolean variable, *load_model*, which, when set to True, loads the model back into the session using *tf.train.Saver.restore()* method, based on the provided *MODEL_NUMBER.*

DDPG Training & Testing: Save Gifs
To visualize our policy changes over time, we save the rendering animation of a single episode for every *NEXT_GIF_COUNT* episode. Within the *Worker.work()* method, we append the RGB arrays obtained using *env.step()* from each step into a list. Using the *mimwrite()* function from the *imageio* library, we create a GIF file and save it to the '*gifs_save'* directory. The file follows the format *'episode_<episode count>_<episode length>_<episode reward>.gif*'.

DDPG Training & Testing: Save Episode Buffer
We also store our episode buffer information in the *'/episode_buffer_save*' directory after every episode. This is achieved using the *pickle* library and the *pickle.dump()* function. The file naming convention is *'model_<episode_count>.dat'.*

Reflection
Throughout this project, our team has gained a deeper understanding of learning agents and their implementation. To make our learning agent work, we made some minor adjustments to our previous gym environment and neural network code, which has been documented in their respective Python file as comments. While our learning agent is capable of performing both training and testing operations, we have yet to train a policy that enables our Aliengo robot to walk smoothly, primarily due to the inefficiency of the training process. To address this, we plan to continue making improvements by adjusting our parameter values. We recognize the need to refine our threshold-based reward structure, such as introducing a relationship with state values like linearity. Furthermore, we are considering expanding our termination states beyond checking collisions between joints. This may include terminating when the robot's position is too low or when the robot is falling backwards, allowing us to skip over undesirable states. All in all, we hope to be able to achieve a working policy for our final submission.